1. Solution:

We want the value of overall chance of the failure, $1 - \prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$, to be as small as possible. Which is equivalent to make $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ as large as possible. So, this question is to find the optimal combination of the number of backup subsystems for every subsystem, and make their cost is less or equal to B. Let $dp[0..B]$ is an array, $dp[i]$ denote that the maximal value of $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ when our budget is i. Then we can get the following recurrence formula:

$dp[i] = (dp[i - c_k]\frac{1-(1-r_k)^{b_k+1}}{1-(1-r_k)^{b_k}})$ When $i - c_k \geq 0$

In order to compute this value, we need to store the selection method of every budget i. That is, for every $dp[i]$, we will have a table $T_i[1..n]$ to store the number of every subsystem. That's where we can get our b_k on the formula above ($b_k = T_i[k]$). Obviously, for $i = 0$, $T_0[1..n] = [0, 0, ..., 0]$ and $dp[0] = \prod_{i=1}^{n}(1 - (1 - r_i)^{0}) = 0$. So, we have initialized our base problem. We can then compute $dp[1]$, $dp[2]$, …, $dp[B]$. Of course, we must also keep track of the value of $T_i[1..n]$. After we compute the $dp[B]$. The $T_B[1..n]$ will record the number of every subsystem that will get a maximal value of $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ (the probability of success). Total running time is $O(Bn)$, because for every i, we need $O(n)$ time to compute the optimal combination and the optimal value $dp[i]$.

2. Solution:

```
Alg:uniquePaths(self, m: int, n: int) -> int:
    d = [[1...1]...[1...1]] # n * m
    for col->1 to m:
        for row->1 to n:
            d[col][row] = d[col - 1][row] + d[col][row - 1]

    return d[m - 1][n - 1]
```

3. Solution:

Suppose we know that a particular item of weight w is in the solution. Then we must solve the subproblem on n − 1 items with maximum weight W − w. Thus, to take a bottom-up approach we must solve the 0-1 knapsack problem for all items and possible weights smaller than W. We'll build an n + 1 by W + 1 table of values where

the rows are indexed by item and the columns are indexed by total weight. (The first row and column of the table will be a dummy row). For row i column j, we decide whether or not it would be advantageous to include item i in the knapsack by comparing the total value of of a knapsack including items 1 through i − 1 with max weight j, and the total

value of including items 1 through i − 1 with max weight j − i.weight and also item i. To solve the problem, we simply examine the n, W entry of the table to determine the maximum value we can achieve. To read off the items we include, start with entry n, W. In general, proceed as follows: if entry i, j equals entry i−1, j, don't include item i, and examine entry i−1, j next. If entry i, j doesn't

equal entry i − 1, j, include item i and examine entry i − 1, j − i.weight next. See algorithm below for construction of table:

Algorithm 1 0-1 Knapsack(n,W)

```
Initialize an n + 1 by W + 1 table K
for j = 1 to W do
    K[0, j] = 0
end for
for i = 1 to n do
    K[i, 0] = 0
end for
for i = 1 to n do
    for j = 1 to W do
        if j < i.weight then
            K[i, j] = K[i − 1, j]
        end if
        K[i, j] = max(K[i − 1, j], K[i − 1, j − i.weight] + i.value)
    end for
end for
```

4.

Many kinds of solutions. Here some examples:

Dp:

```java
enum Index {
    GOOD, BAD, UNKNOWN
}

public class Solution {
    Index[] memo;

    public boolean canJumpFromPosition(int position, int[] nums) {
        if (memo[position] != Index.UNKNOWN) {
            return memo[position] == Index.GOOD ? true : false;
        }

        int furthestJump = Math.min(position + nums[position], nums.length - 1);
        for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition++) {
            if (canJumpFromPosition(nextPosition, nums)) {
                memo[position] = Index.GOOD;
                return true;
            }
        }

        memo[position] = Index.BAD;
        return false;
    }

    public boolean canJump(int[] nums) {
        memo = new Index[nums.length];
        for (int i = 0; i < memo.length; i++) {
            memo[i] = Index.UNKNOWN;
        }
        memo[memo.length - 1] = Index.GOOD;
        return canJumpFromPosition(0, nums);
    }
}
```

Dp-bottom-up

```java
enum Index {
    GOOD, BAD, UNKNOWN
}

public class Solution {
    public boolean canJump(int[] nums) {
        Index[] memo = new Index[nums.length];
        for (int i = 0; i < memo.length; i++) {
            memo[i] = Index.UNKNOWN;
        }
        memo[memo.length - 1] = Index.GOOD;

        for (int i = nums.length - 2; i >= 0; i--) {
            int furthestJump = Math.min(i + nums[i], nums.length - 1);
            for (int j = i + 1; j <= furthestJump; j++) {
                if (memo[j] == Index.GOOD) {
                    memo[i] = Index.GOOD;
                    break;
                }
            }
        }

        return memo[0] == Index.GOOD;
    }
}
```

Greedy:

```java
public class Solution {
    public boolean canJump(int[] nums) {
        int lastPos = nums.length - 1;
        for (int i = nums.length - 1; i >= 0; i--) {
            if (i + nums[i] >= lastPos) {
                lastPos = i;
            }
        }
        return lastPos == 0;
    }
}
```